3.7 IMP++: IMP Extended with Several Features

Our goal in the remaining of this chapter is to discuss the advantages and disadvantages of the various language operational semantics styles in more depth. To have a basis for the discussion, we extend the IMP language discussed previously in this chapter with several simple language features and attempt to give the extended language, called IMP++, an operational semantics in each of the discussed operational semantics styles. The additional features of IMP++ are the following:

1. A variable increment operation, `++` *Var*, whose role is to infuse side effects in expressions;

2. An output statement, `output(`*AExp*`)`, whose role is to modify the configurations (one needs to add an output "buffer", where all the output is collected);

3. Abrupt termination, both by means of an explicit `halt` statement and by means of division-by-zero, whose role is to enforce a sudden change of the evaluation context; and

4. Spawning a new thread, `spawn(`*Stmt*`)`, which executes the stement given as argument concurrently with the rest of the program, sharing all the variables. The role of the `spawn` statement is to test the support of the various operational semantics for concurrent language features.

The criterion used for selecting these new features of IMP++ was twofold: on the one hand, these are quite ordinary features encountered in many languages; on the other hand, they incrementally expose the limitations of the various operational semantics styles. Both IMP and IMP++ are admittedly toy languages; however, if a certain programming language operational semantics style has difficulties in supporting any of the features of IMP or any of the above IMP extensions in IMP++, or if in order to define a new feature one needs to make unrelated changes in the already existing semantics of other features, then one should most likely expect the same problems, but of course amplified, to occur in practical attempts to define real-life programming languages.

IMP++ extends IMP both syntactically and semantically. Syntactically, it adds to IMP the following four language constructs:

$$\begin{array}{rcl} \mathit{AExp} & ::= & \ldots \mid \texttt{++}\ \mathit{Var} \\ \mathit{Stmt} & ::= & \ldots \mid \texttt{output(}\mathit{AExp}\texttt{)} \mid \texttt{halt} \mid \texttt{spawn(}\mathit{Stmt}\texttt{)} \end{array}$$

Semantically, in addition to defining the new language constructs above, we prefer that division-by-zero implicitly halts the program in IMP++, same like the explicit use of `halt`, but in the middle of an expression evaluation. When such an error takes place, one could also generate an error message; however, for simplicity, we do not consider error messages here.

Before we continue with the details of defining each of the new language features in each of the semantics, we would like to mention that there could be various ways to do these. Our goal in this section is to illustrate the lack of modularity of the various semantic styles in different language extension scenarios, and not necessarily to output the user good error messages. For example, a program that performs a division by zero simply halts its execution when the division by zero takes place and all is reported is the program state and output at that moment. One could certainly envision better ways to terminate the program; we encourage the reader to experiment with that in particular and, in general, to try to improve upon our definitions below wherever possible.

3.7.1 Adding Variable Increment

Like in several main-stream programming languages, "`++ x`" increments the value of `x` in the state and evaluates to the incremented value. This way, the increment operation makes the evaluation of expressions to now have side effects.

Big-Step Operational Semantics

The big-step operational semantics is the most affected by the inclusion of side effects in expressions, because the previous triples $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ need to change to four-tuples of the form $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$, to account for collecting the possible side effects generated by the evaluation of expressions (note that the evaluation of boolean expressions, because of `<=`, can also have side effects). The big-step semantics of all the language constructs needs to change as well. For example, the semantics of `/` changes as follows:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1\ /\ a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2\rangle}\ ,\quad \text{where } i_2 \neq 0 \qquad (\textsc{BigStep-Div-Left-to-Right})$$

$$\frac{\langle a_1, \sigma_2\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1\ /\ a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_1\rangle}\ ,\quad \text{where } i_2 \neq 0 \qquad (\textsc{BigStep-Div-Right-to-Left})$$

The rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. We will shortly explain why they fail to fully capture the desired behaviors.

Let us next include the big-step semantics of the increment operation; once all the changes to the existing semantics of IMP are applied, the big-step semantics of increment is straightforward:

$$\langle \texttt{++}\, x, \sigma\rangle \Downarrow \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle \qquad (\textsc{BigStep-Inc})$$

Indeed, the problem with big-step is not to define the semantics of variable increment, but what it takes to be able to do it. One needs to redefine configurations as explained above and, consequently, to change the semantics of almost all the already existing features of IMP to use the new configurations. This, and other features defined later on, show how non-modular big-step semantics is.

Moreover, the addition of side-effects makes the originally intended evaluation strategies of the various expression constructs important. Indeed, for demonstration purposes we originally wanted `+` and `/` to be non-deterministic (i.e., to evaluate their arguments stepwise non-deterministically, possibly interleaving their evaluations), while `<=` to be left-right sequential. These different evaluation strategies can now lead to different behaviors, which, unfortunately, cannot be naturally captured by the big-step, or "natural" semantics. While we can still capture some limited degree of non-determinism as we showed above with the definition of `/`, namely we can choose which of the sub-expressions to evaluate first, we cannot define the full non-deterministic strategy (unless we make radical changes to the definition, such as working with sets of values instead of values, which significantly complicates the big-step definition and still fails to capture the non-deterministic behaviors — it would only capture the non-deterministic evaluation results). To see how the non-deterministic "choice" evaluation strategy in big-step semantics fails to capture all the desired behaviors, consider the the expression "`++ x / (++ x / x)`" with `x` initially 1. This expression can only evaluate to 1, 2 or 3 under non-deterministic choice strategy like we get in big-step, but it

can also evaluate to 0 and even perform a division-by-zero under a fully non-deterministic evaluation strategy, like we will correctly get using the other operational semantic approaches.

We would like to re-emphasize that big-step semantics not only misses behaviors due to its lack of support for non-deterministic evaluation strategies, like shown above, but also hides misbehaviors that it, in principle, detects. For example, the expression "`1 / (x / ++ x)`" (assume `x` > 0) can either evaluate to 1 or perform an erroneous division by zero. If one searches for all the possible evaluations of a program containing such an expression using the big-step semantics in this section, one will only see the behavior where this expression evaluates to 1; one will never see the erroneous behavior where the division by zero takes place. This will be fixed shortly when we modify the big-step semantics to support abrupt termination. However, without modifying the semantics, the unfortunate language designer using big-step semantics may wrongly think that the program is correct. Contrast that with the small-step semantics, which, even when one does not add support for abrupt termination, still detects the wrong behavior by getting stuck on the configuration obtained right before the division by zero.

Additionally, as explained in Section 3.2.3, the new configurations may also interfere with the rewriting infrastructure when one wants to execute big-step definitions using rewriting. Indeed, one needs to remove resulting rules that lead to non-termination, such as the rewriting rules of the form $R \to R$ corresponding to big-step sequents $R \Downarrow R$ where R are result configurations (e.g., $\langle i, \sigma \rangle$ with $i \in \mathit{Int}$ or $\langle t, \sigma \rangle$ with $t \in \{\texttt{true}, \texttt{false}\}$). Of course, we do not use this argument against big-step operational semantics (its serious break of modularity is already sufficient to disqualify big-step in the competition for an ideal language definitional framework), but rather as a warning message to the interested reader who wants to use rewriting logic to define and possibly to use rewriting engines (like Maude) to execute big-step operational semantics.

Exercise 31. *Add variable increment to* IMP*, using big-step semantics:*

1. *Write the complete big-step operational semantics as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.9;*
3. ☆ *Implement the resulting rewrite logic theory in Maude, like in Figure 3.10. To test it, add to the IMP programs in Figure 3.4 the following two:*

```
ops nondet sum++ : -> Stmt .

eq sum++ = (
     s := 0 ; m := 0 ;
     while (++ m <= n) do (s := s + m)
) .

eq nondet = (
     x := 1 ;
     x := ++ x / (++ x / x)
) .
```

The first program should have only one behavior, so, for example, both Maude commands below

```
Maude> rewrite < sum++, n |-> 100 > .
Maude> search  < sum++, n |-> 100 > =>! Cfg:Configuration .
```

should show the same result configuration, `< m |-> 101 , n |-> 100 , s |-> 5050 >`*. The second program should have (only) three different behaviors under big-step semantics; the first command below will show one of the three behaviors, but the second will show all three of them:*

```
rewrite < nondet, empty > .
search  < nondet, empty > =>! Cfg:Configuration .
```

The three behaviors captured by the big-step semantics result in configurations `< x |-> 1 >`*,* `< x |-> 2 >`*, and* `< x |-> 3 >`*. As explained above, the big-step semantics so far should not be able to expose the behaviors in which* `x` *is* `0` *and in which a division-by-zero takes place.*

Small-Step Operational Semantics

Including side effects in expressions is not as bad in small-step semantics as in big-step semantics, because, as discussed in Section 3.3, in small-step operational semantics one typically includes all the configuration components in each configuration; thus, expressions, like any other syntactic categories including statements, can seamlessly modify the state if they need to. However, since we deliberately did not anticipate the inclusion of side effects in expression evaluation, we still have to go back through the existing definition and modify *all* the expression rules to propagate the side effects. For example, the small-step rule

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \to \langle a_1' \ / \ a_2, \sigma\rangle}$$

for the first argument of `/` does not apply anymore when the next step in a_1 is an increment operation (since the state σ changes), so it needs to change to

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma_1\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \to \langle a_1' \ / \ a_2, \sigma_1\rangle}$$

Of course, all these changes due to side-effect propagation would have not been necessary if we anticipated that side effects may be added to the language, but the entire point of this exercise is to study the strengths of the various semantic approaches without knowing what comes next.

Once all the changes are applied, one can define the small-step semantics of the increment operation almost identically to its big-step semantics:

$$\langle \texttt{++}\, x, \sigma\rangle \to \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle \qquad \text{(SmallStep-Inc)}$$

Exercise 32. *Same as Exercise 31, but for small-step instead of big-step.* ☆ *Make sure that the small-step definition in Maude exhibits all five behaviors of program* `nondet` *defined in Exercise 31 (the three behaviors exposed by the big-step definition in Maude in Exercise 31, plus one where* `x` *is* `0` *and one where a division by zero is taking place).*

Modular Structural Operational Semantics

In MSOS, one can define the increment modularly:

$$\texttt{++}\,x \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[(\sigma_0(x)+_{Int}1)/x],\ldots\}} \sigma_0(x) +_{Int} 1 \qquad \text{(MSOS-Inc)}$$

No other rule needs to be changed, because MSOS already assumes that, unless otherwise specified, each rule propagates all the configuration changes in its condition(s).

Exercise 33. *Same as Exercise 31, but for MSOS instead of big-step.* ☆ *Make sure that the MSOS definition in Maude, like the small-step definition in Maude in Exercise 32, exhibits all five behaviors of program* `nondet` *defined in Exercise 31.*

Reduction Semantics with Evaluation Contexts

Evaluation contexts can also be elegantly used to define increment modularly:

$$\langle c, \sigma\rangle[\texttt{++}\,x] \to \langle c, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle[\sigma(x) +_{Int} 1] \qquad \text{(CxtRed-Inc)}$$

No other rule needs to change, because the language-specific rules of a context reduction definition are unconditional, each rule matching and modifying only its related part of the configuration. In other words, each rule application propagates all the configuration changes due to the applications of other rules. Note that the only conditional rule, the characteristic rule of context reduction which is language-independent and says that a reduction can be applied in any appropriate context, does not inhibit the propagation of side-effects either.

Exercise 34. *Same as Exercise 31, but for reduction semantics with evaluation contexts instead of big-step.* ☆ *Like for the Maude definitions of small-step and MSOS above, make sure that the resulting Maude definition exhibits all five behaviors of program* `nondet` *defined in Exercise 31.*

The Chemical Abstract Machine

The chemical abstract machine can also define the increment modularly:

$$\{\!|\texttt{++}\,x \triangleright c|\!\}, \{\!|x \mapsto i \triangleright \sigma|\!\} \to \{\!|i +_{Int} 1 \triangleright c|\!\}, \{\!|x \mapsto i +_{Int} 1 \triangleright \sigma|\!\} \qquad \text{(CHAM-Inc)}$$

Exercise 35. *Same as Exercise 31, but for CHAM instead of big-step.* ☆ *Like for the Maude definitions of small-step, MSOS and reduction semantics with evaluation contexts above, make sure that the resulting Maude definition exhibits all five behaviors of program* `nondet` *in Exercise 31.*

3.7.2 Adding Output

The semantics of the output statement `output(`a`)` is that a is first evaluated to some integer i, which is then collected in an "output buffer". By an output buffer we here mean some list structure to which one can add more elements; removing of elements from that list is not allowed. In a formal language semantics, collecting the output in a list is acceptable; in implementations of the language, one will most likely want the output to be displayed as it is generated. Let us therefore assume comma-separated integer lists, say $\mathsf{List}[Int]$, and let ω, ω', ω_1, etc., range over such lists of integers.

There is some flexibility as to where the output buffer should be located in the configuration. One possibility is as a new top-level component in the configuration. Another possibility is as a special variable in the already existing state. The latter would require some non-trivial changes in the mathematical model of the state, so we prefer to follow the former approach in what follows. An additional argument for our choice is that sooner or later one needs to add new components to the configuration anyway, so we take this opportunity to discuss how robust/modular the various semantic styles are with regards to changes in the structure of the configuration.

Big-Step Operational Semantics

To accommodate the output buffer, the big-step result configurations for statements need to change from state configurations $\langle \sigma \rangle$ with $\sigma \in \mathit{State}$, to pairs $\langle \sigma, \omega \rangle$, where $\omega \in \mathsf{List}[\mathit{Int}]$. That means, in particular, that the big-step semantics of all the statements implicitly needs to change, to accommodate the additional configuration output component. For example, the semantics of sequential composition needs to change from

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle,\ \langle s_2, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle}{\langle s_1 \,;\, s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle}$$

to

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1, \omega_1 \rangle,\ \langle s_2, \sigma_1 \rangle \Downarrow \langle \sigma_2, \omega_2 \rangle}{\langle s_1 \,;\, s_2, \sigma \rangle \Downarrow \langle \sigma_2, (\omega_1, \omega_2) \rangle}$$

These necessary changes in big-step semantics show, again, the lack of modularity of big-step semantics. Once all the changes are applied to "update" the existing semantics to the new configurations, one can easily define the semantics of the output statement as follows:

$$\frac{\langle a, \sigma \rangle \Downarrow \langle i \rangle}{\langle \texttt{output(}a\texttt{)}, \sigma \rangle \Downarrow \langle \sigma, i \rangle} \qquad \text{(BigStep-Output)}$$

Exercise 36. *Add output to the current* IMP++ *language (i.e.,* IMP *with increment), using big-step semantics:*

1. *Write the complete big-step operational semantics as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.9;*
3. ☆ *Implement the resulting rewrite logic theory in Maude, like in Figure 3.10. To test it, modify the* `sum++` *example in Exercise 31 as follows:*

```
s := 0 ; m := 0 ;
while (++ m <= n) do (s := s + m ; output m)
```

Small-Step Operational Semantics

While only some of the configurations of big-step semantics had to change to accommodate the output statement, in the case of small-step semantics *all configurations need to change*, so, implicitly, the small-step semantics of all the statements needs to change. The changes are straightforward,

essentially having to just propagate the output through each language construct, but they are still changes and thus expose, again, the lack of modularity of small-step operational semantics. Here is, for example, how the small-step rule for the first argument of / needs to change (this is the second time this rule changes; the first time it changed when we added the increment):

$$\frac{\langle a_1, \sigma, \omega\rangle \to \langle a_1', \sigma_1, \omega\rangle}{\langle a_1 \ / \ a_2, \sigma, \omega\rangle \to \langle a_1' \ / \ a_2, \sigma_1, \omega\rangle}$$

We again assumed just the current design of the language, without attempting to anticipate other features that will be possibly added in the future. For example, if functions will be added to the language, in which case expressions will also possibly affect the output through function calls, then the rule above will need to change again, to allow for the expressions to modify the output.

Once all the changes are applied, one can give the small-step semantics of output as follows:

$$\frac{\langle a, \sigma, \omega\rangle \to \langle a', \sigma', \omega\rangle}{\langle \texttt{output}(a), \sigma, \omega\rangle \to \langle \texttt{output}(a'), \sigma', \omega\rangle} \qquad \text{(SmallStep-Output-Arg)}$$

$$\langle \texttt{output}(i), \sigma, \omega\rangle \to \langle \texttt{skip}, \sigma, (\omega, i)\rangle \qquad \text{(SmallStep-Output)}$$

Exercise 37. *Same as Exercise 36, but for small-step instead of big-step.*

Modular Structural Operational Semantics

MSOS can modularly support the output extension of the language. All one needs to do is to add a new write-only attribute *output* in the label for collecting the output and then to add the rules for the output statement as follows:

$$\frac{a \xrightarrow{\mathcal{L}} a'}{\texttt{output}(a) \xrightarrow{\mathcal{L}} \texttt{output}(a')} \qquad \text{(MSOS-Output-Arg)}$$

$$\texttt{output}(i) \xrightarrow{\{output'=i,\ldots\}} \texttt{skip} \qquad \text{(MSOS-Output)}$$

Note that, since *output* is a write-only attribute, we only need to mention the new value that is added to the output in the label of the second rule above. If *output* was declared as a read-write attribute, then the label of the second rule above would have been $\{output = \omega, output' = (\omega, i), \ldots\}$. Recall that an aim of MSOS is to minimize the amount of information that the user needs to write in each rule. Indeed, anything written by a user can lead to non-modularity and thus work against the user when changes are performed to the language; for example, if for some reason one declared *output* as a read-write attribute and then later on one decided to change the list construct for the output integer list from comma "_, _" to something else, say "_ : _", then one would need to change the label in the second rule above from $\{output = \omega, output' = (\omega, i), \ldots\}$ to $\{output = \omega, output' = (\omega : i), \ldots\}$.

Exercise 38. *Same as Exercise 36, but for MSOS instead of big-step.*

Reduction Semantics with Evaluation Contexts

One needs to first change the context reduction configuration from $\langle s, \sigma \rangle$ to $\langle s, \sigma, \omega \rangle$, to also include the output. This change, unfortunately, generates several other changes in the existing semantics, some of them non-modular in nature. First, one needs to change the syntax of contexts from $\mathit{Cxt} ::= ... \mid \langle \mathit{Cxt}, \mathit{State} \rangle$ to $\mathit{Cxt} ::= ... \mid \langle \mathit{Cxt}, \mathit{State}, \mathsf{List}[\mathit{Int}] \rangle$; some change in the configuration is unavoidable in any approach (even in MSOS we added a new label), so this is not problematic. The problematic changes are the following. The rules for variable lookup and for assignment needed the complete configuration, so they need to change from

$$\langle c, \sigma \rangle[x] \to \langle c, \sigma \rangle[\sigma(x)]$$
$$\langle c, \sigma \rangle[x := i] \to \langle c, \sigma[i/x] \rangle[\texttt{skip}]$$
$$\langle c, \sigma \rangle[\texttt{++}\, x] \to \langle c, \sigma[(\sigma(x) +_{\mathit{Int}} 1)/x] \rangle[\sigma(x) +_{\mathit{Int}} 1]$$

to

$$\langle c, \sigma, \omega \rangle[x] \to \langle c, \sigma, \omega \rangle[\sigma(x)]$$
$$\langle c, \sigma, \omega \rangle[x := i] \to \langle c, \sigma[i/x], \omega \rangle[\texttt{skip}]$$
$$\langle c, \sigma, \omega \rangle[\texttt{++}\, x] \to \langle c, \sigma[(\sigma(x) +_{\mathit{Int}} 1)/x], \omega \rangle[\sigma(x) +_{\mathit{Int}} 1]$$

Once the changes are applied, one can add the evaluation context for output as well as its context reduction rule as follows:

$$\mathit{Cxt} ::= ... \mid \texttt{output(}\mathit{Cxt}\texttt{)}$$
$$\langle c, \sigma, \omega \rangle[\texttt{output(}i\texttt{)}\,] \to \langle c, \sigma, (\omega, i) \rangle[\texttt{skip}] \qquad \text{(CxtRed-Output)}$$

Other possibilities to add the output to the configuration and to give the context reduction semantics of the above language features in a way that appears to be more modular (but which yields other problems) are discussed in Section 3.8.

Exercise 39. *Same as Exercise 36, but for reduction semantics with evaluation contexts instead of big-step.*

The Chemical Abstract Machine

All we have to do is to add a new output molecule in the top-level solution that holds the output as a list. Then to include the output statement in the semantics we first need to give its evaluation strategy by means of a heating/cooling pair and then the reaction rule that dissolves the output statement and collects its argument into the output molecule via an airlock followed by a cooling of the output molecule:

$$\{\!|\texttt{output(}a\texttt{)} \triangleright c|\!\} \rightleftharpoons \{\!|a \triangleright \{\!|\texttt{output(}\,\square\,\texttt{)} \triangleright c|\!\}|\!\}$$
$$\{\!|\texttt{output(}i\texttt{)} \triangleright c|\!\},\ \{\!|w|\!\} \to \{\!|\texttt{skip} \triangleright c|\!\},\ \{\!|i \triangleright \{\!|w|\!\}|\!\} \qquad \text{(CHAM-Output)}$$
$$\{\!|i \triangleright \{\!|w|\!\}|\!\} \rightharpoonup \{\!|w, i|\!\} \qquad \text{(CHAM-Cool-Output-Molecule)}$$

We cannot drop the last cooling rule and replace `(CHAM-Output)` with "$\{\!|\texttt{output(}i\texttt{)} \triangleright c|\!\},\ \{\!|w|\!\} \to \{\!|\texttt{skip} \triangleright c|\!\},\ \{\!|w, i|\!\}$" because this rule would violate the basic CHAM "chemical" assumption that one can only use the airlock operation to pull/insert items in a solution molecule (of the form $\{\!|...|\!\}$).

Exercise 40. *Same as Exercise 36, but for the CHAM instead of big-step.*

3.7.3 Adding Abrupt Termination

As discussed, we add both implicit and explicit abrupt program termination in IMP++. The implicit abrupt termination is given by division by zero, while the explicit abrupt termination is given by a new statement added to the language, `halt`.

For the sake of making a choice and also for demonstration purposes, in both cases of abrupt termination we would like, admittedly subjectively, the resulting configuration to have the same structure as if the program terminated normally; for example, in the case of big-step semantics, we would like the result configuration for statements to be a pair $\langle \sigma, \omega \rangle$, where s is the state and ω is the output when the program was terminated abruptly. Unfortunately, that is not possible in all cases without having to intrusively modify the syntax of the IMP language (to "catch" the exceptional behavior and explicitly discard the additional information), since some operational styles need to make a sharp distinction between a halting configuration and a normal configuration (for propagation reasons). Adepts of those semantic styles may argue that our semantic choice above seems inappropriate, since giving more information in the result configuration, such as "this is a halting configurations", is better for all purposes than giving less information. There are, however, also reasons to always want a normal result configuration upon termination. For example, one may want to include IMP in a larger context, such as in a distributed system, where all the context wants to know about the embedded language is that it takes a statement and produces a state; IMP's internal exceptional situations are of no concern to the outer context.

We believe that there is no absolute better or worse language design, both in what regards syntax and in what regards semantics. Our task here is to make the language designer aware of the subtleties and the limitations of the various semantic approaches.

Big-Step Operational Semantics

The lack of modularity of big-step semantics will be, again, emphasized here. Let us first add the semantic definition for the implicit abrupt termination generated by division by zero. Recall that one of the (modified for increment) big-step semantics rules for division was the following:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1, \sigma_1 \rangle,\ \langle a_2, \sigma_1 \rangle \Downarrow \langle i_2, \sigma_2 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle i_1 \ /_{Int} \ i_2, \sigma_2 \rangle} \text{, where } i_2 \neq 0$$

We keep that unchanged, but we also add the new rule:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1, \sigma_1 \rangle,\ \langle a_2, \sigma_1 \rangle \Downarrow \langle 0, \sigma_2 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle \texttt{error}, \sigma_2 \rangle} \qquad \text{(BigStep-Div-By-Zero-Left-To-Right)}$$

In the above rule, `error` can be regarded as a special value; alternatively, one can regard $\langle \texttt{error}, \_ \rangle$ as a special result configuration holding only one state. Note that we cannot discard the state, because we want to include both the state and the output in the final configuration when the program terminates abruptly.

But what if the evaluation of a_1 or of a_2 in the above rule generates itself an error? If that is the case, then one needs to propagate that error through the division construct:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle \texttt{error}, \sigma_1 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle \texttt{error}, \sigma_1 \rangle} \qquad \text{(BigStep-Div-Left-to-Right-Error-Left)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma_1\rangle \Downarrow \langle \texttt{error}, \sigma_2\rangle}{\langle a_1\ \texttt{/}\ a_2, \sigma\rangle \Downarrow \langle \texttt{error}, \sigma_2\rangle}\ \text{(BigStep-Div-Left-to-Right-Error-Right)}$$

Note that in case a_1 generates the error then a_2 is not even evaluated anymore, to faithfully capture the intended abrupt termination. To maximally capture the evaluation non-determinism of `/` in big-step semantics, we need to add two other similar rules for the left-to-right evaluation of `/`.

Unfortunately, one has to do the same for all the expression language constructs. This way, for each expression construct, one has to add at least as many error-propagation big-step rules as arguments that expression construct takes. Moreover, when the evaluation error reaches a statement, one needs to transform it into a "halting signal". This can be achieved by introducing a new type of result configuration, namely $\langle \texttt{halt}, s, \omega\rangle$, and then adding appropriate halting propagation rules for all the statements. For example, the assignment statement needs to be added the new rule

$$\frac{\langle a, \sigma\rangle \Downarrow \langle \texttt{error}, \sigma'\rangle}{\langle x\ \texttt{:=}\ a, \sigma\rangle \Downarrow \langle \texttt{halt}, \sigma', \epsilon\rangle}\ \text{(BigStep-Asgn-Halt)}$$

The halting signal needs to be propagated through statement constructs, collecting the appropriate state and output. For example, the following two rules need to be included for sequential composition, in addition to the existing rule (which stays unchanged):

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \texttt{halt}, \sigma_1, \omega_1\rangle}{\langle s_1\ \texttt{;}\ s_2, \sigma\rangle \Downarrow \langle \texttt{halt}, \sigma_1, \omega_1\rangle}\ \text{(BigStep-Seq-Halt-Left)}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma_1, \omega_1\rangle,\ \langle s_2, \sigma_1\rangle \Downarrow \langle \texttt{halt}, \sigma_2, \omega_2\rangle}{\langle s_1\ \texttt{;}\ s_2, \sigma\rangle \Downarrow \langle \texttt{halt}, \sigma_2, (\omega_1, \omega_2)\rangle}\ \text{(BigStep-Seq-Halt-Right)}$$

In addition to all the halting propagation rules, we also have to define the semantics of the explicit halt statement:

$$\langle \texttt{halt}, \sigma\rangle \Downarrow \langle \texttt{halt}, \sigma, \epsilon\rangle\ \text{(BigStep-Halt)}$$

Therefore, when using big-step semantics, one has to more than *double* the number of rules in order to support abrupt termination. Indeed, any argument of any language construct can yield the termination signal, so a rule is necessary to propagate that signal through the current language construct. It is hard to imagine anything worse in a language design framework. An unfortunate language designer choosing big-step semantics as her favorite language definition framework will incrementally become very reluctant to add or experiment with any new feature in her language. For example, imagine that one wants to add exceptions and break/continue of loops to IMP++.

Finally, unless one extends the language syntax, there appears to be no way to get rid of the junk result configurations $\langle \texttt{halt}, \sigma, \omega\rangle$ that have been artificially added in order to propagate the error or the halting signals. For example, one cannot simply add the rule

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \texttt{halt}, \sigma', \omega\rangle}{\langle s, \sigma\rangle \Downarrow \langle \sigma', \omega\rangle}$$

because it may interfere with other rules and thus wrongly hide the halting signal; for example, it can be applied on the second hypothesis of the rule (BigStep-Seq-Halt-Right) above hiding the halting signal and thus wrongly making the normal rule (BigStep-Seq) applicable. While

having junk result configurations of the form $\langle \texttt{halt}, \sigma, \omega \rangle$ may seem acceptable in our scenario here, perhaps even desirable for debugging reasons, in general one may find it inconvenient to have many types of result configurations; indeed, one would need similar junk configurations for exceptions, for break/continue of loops, for functions return, etc.

One can easily dissolve the junk configurations if one is willing to change the language syntax to know when one is done with the evaluation of the entire program. For example, one can add a "program" construct

$$\mathit{Pgm} ::= \texttt{pgm}\ \mathit{Stmt}$$

together with an appropriate configuration construct and together with the rules

$$\frac{\langle s, \sigma \rangle \Downarrow \langle \sigma', \omega \rangle}{\langle \texttt{pgm}\ s, \sigma \rangle \Downarrow \langle \sigma', \omega \rangle} \qquad \text{(BigStep-Pgm-Normal)}$$

$$\frac{\langle s, \sigma \rangle \Downarrow \langle \texttt{halt}, \sigma', \omega \rangle}{\langle \texttt{pgm}\ s, \sigma \rangle \Downarrow \langle \sigma', \omega \rangle} \qquad \text{(BigStep-Pgm-Halt)}$$

If one does not like to add a new syntactic category (which implies adding also a new configuration construct), then one can have `pgm` defined as a statement construct (but in that case one should be aware of the fact that the pair `pgm`/`halt` acts more like an exception throwing and catching).

In addition to the lack of modularity due to having to more than double the number of rules in order to add abrupt termination, the addition of all these rules can also have a significant impact on performance when one wants to execute the big-step operational semantics. Indeed, there are now seven rules for division, each having the same left-hand side, $\langle a_1 \ / \ a_2, \sigma, \omega \rangle$, and some of these rules even sharing some of the hypotheses. That means that any general-purpose proof or rewrite system attempting to execute such a definition will unavoidably face the problem of searching a large space of possibilities in order to find one or all possible reductions.

Exercise 41. *Add abrupt termination as discussed above to the current* IMP++ *language (i.e.,* IMP *with increment and output), using big-step semantics:*

1. *Write the complete big-step operational semantics as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.9;*
3. ☆ *Implement the resulting rewrite logic theory in Maude, like in Figure 3.10. To test it, modify the* `sum++` *example in Exercise 31 as follows:*

```
s := 0 ; m := 0 ;
while (true) do
  if (++ m <= n) then (
    s := s + m ;
    output m
  ) else halt
```

The resulting definition may be very slow when executed in Maude, even for small values of `n` *(such as 2,3,4) which is normal (as explained above, the search space is now much larger). Unlike in previous extensions of* IMP*, the big-step semantics should now be able to detect the division-by-zero behavior in the program* `nondet` *in Exercise 31.*

Small-Step Operational Semantics

Small-step operational semantics turns out to be almost as non-modular as big-step when it gets to defining control-intensive statements like abrupt termination. Like for big step, we have to invent new "error" expression configurations of the form $\langle \texttt{error}, \sigma, \omega \rangle$ to signal that a division by zero just took place. With that, we can then define the small-step semantics of division by zero as follows (recall that the original SmallStep-Div rule was "$\langle i_1 \texttt{ / } i_2, \sigma \rangle \to \langle i_1 /_{Int} i_2, \sigma \rangle$ where $i_2 \neq 0$"):

$$\langle i_1 \texttt{ / } 0, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle \qquad \text{(SmallStep-Div-By-Zero)}$$

Like for the big-step semantics, we have to make sure that the error signal is correctly propagated. Here are, for example, the propagation rules through the division construct:

$$\frac{\langle a_1, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle}{\langle a_1 \texttt{ / } a_2, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle} \qquad \text{(SmallStep-Div-Error-Left)}$$

$$\frac{\langle a_2, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle}{\langle a_1 \texttt{ / } a_2, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle} \qquad \text{(SmallStep-Div-Error-Right)}$$

The two rules above are given in such a way that the semantics is faithful to the intended *computational granularity* of the defined language feature. Indeed, we want division by zero to take one computational step to be reported as an error, as opposed to as many steps as the depth of the context in which the error has been detected; for example, an expression like `(3 / 0) / 3` should reduce to `error` in one step, not in two. If one added `error` as a special integer value and replaced the two rules above by

$$\langle \texttt{error} \texttt{ / } a_2, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle$$

$$\langle a_1 \texttt{ / error}, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle$$

then errors would be propagated to the top level of the program in as many small-steps as the depth of the context in which the error was generated; we do not want that.

Like in the big-step semantics, the implicit expression errors need to propagate through the statements and halt the program. Unlike for big-step, we can reuse the already existing statement configurations for that purpose; all we have to do is to generate an explicit halt statement, like we exemplify in the following rule for propagating errors through the assignment construct:

$$\frac{\langle a, \sigma, \omega \rangle \to \langle \texttt{error}, \sigma, \omega \rangle}{\langle x \texttt{ := } a, \sigma, \omega \rangle \to \langle \texttt{halt}, \sigma, \omega \rangle} \qquad \text{(SmallStep-Asgn-Halt)}$$

Like for the propagation of errors above, we also want to maintain the intended computational granularity of the halt propagation, namely that the execution of the program should stop as soon as the halting signal has been encountered and *not* in as many steps as the depth of the execution context where the halting signal has been generated. That means, in particular, that IMP's small-step rule for sequential composition needs to change (!) as follows, pointing out an additional source of non-modularity in small-step operational semantics:

$$\frac{\langle s_1, \sigma, \omega \rangle \to \langle s_1', \sigma', \omega' \rangle}{\langle s_1 \texttt{ ; } s_2, \sigma, \omega \rangle \to \langle s_1' \texttt{ ; } s_2, \sigma', \omega' \rangle} \quad , \quad \text{where } s_1' \neq \texttt{halt} \qquad \text{(SmallStep-Seq-Modified)}$$

When $s_1' = \texttt{halt}$ or when s_1 is already `halt` we want to halt the sequential composition right away:

$$\frac{\langle s_1, \sigma, \omega\rangle \to \langle \texttt{halt}, \sigma, \omega\rangle}{\langle s_1 \,\texttt{;}\, s_2, \sigma, \omega\rangle \to \langle \texttt{halt}, \sigma, \omega\rangle} \qquad \text{(SmallStep-Seq-Halt1)}$$

$$\langle \texttt{halt}\,\texttt{;}\, s_2, \sigma, \omega\rangle \to \langle \texttt{halt}, \sigma, \omega\rangle \qquad \text{(SmallStep-Seq-Halt2)}$$

Propagating the halting signal through all the language constructs means that an abruptly terminated program will end up with a result configuration of the form $\langle \texttt{halt}, \sigma, \omega\rangle$. Like in the big-step operational semantics of abrupt termination above, there seems to be no immediate way to terminate (in a small-step) the program with a normal result configuration, namely one of the form $\langle \texttt{skip}, \sigma, \omega\rangle$, both when the program terminates abruptly and when it does not. Adding the rule

$$\langle \texttt{halt}, \sigma, \omega\rangle \to \langle \texttt{skip}, \sigma, \omega\rangle$$

does not work, because it requires an additional step thus violating the intended computational granularity of `halt`; additionally, it may interfere "unexpectedly" with rules like (SmallStep-Seq-Modified) (when $s_1 = \texttt{halt}$), so those rules would need to be changed as well.

To terminate the program with a normal result configuration, we can adopt the same syntax change as for the big-step semantics, namely to add a "`pgm` *Stmt*" construct together with the rules

$$\frac{\langle s, \sigma, \omega\rangle \to \langle s', \sigma', \omega'\rangle}{\langle \texttt{pgm}\ s, \sigma, \omega\rangle \to \langle \texttt{pgm}\ s', \sigma', \omega'\rangle} \text{ , where } s' \neq \texttt{halt} \qquad \text{(SmallStep-Pgm-Normal)}$$

$$\frac{\langle s, \sigma, \omega\rangle \to \langle \texttt{halt}, \sigma, \omega\rangle}{\langle \texttt{pgm}\ s, \sigma, \omega\rangle \to \langle \texttt{pgm skip}, \sigma, \omega\rangle} \qquad \text{(SmallStep-Pgm-Halt1)}$$

$$\langle \texttt{pgm halt}, \sigma, \omega\rangle \to \langle \texttt{pgm skip}, \sigma, \omega\rangle \qquad \text{(SmallStep-Pgm-Halt2)}$$

The rules above are designed such that no computational step is wasted when a halting signal takes place: the whole program evaluates to "`pgm skip`" within the same step. Note that, like for big-step semantics, a new type of configuration is needed to hold programs and that the result program configurations have the form $\langle \texttt{pgm skip}, \sigma, \omega\rangle$.

Exercise 42. *Same as Exercise 41, but for small-step instead of big-step.*

Modular Structural Operational Semantics

MSOS allows for a modular semantics of abrupt termination but, unfortunately, in order to do so it needs to extend the syntax of the language with a construct like `pgm` above. The idea is to use the flexible labels mechanism of MSOS to carry the information that a configuration is in a halting status. Let us assume an additional write-only boolean attribute in the MSOS labels, called *halting*, which is *yes* whenever the program needs to halt. Then we can add the following two natural MSOS rules that set the *halting* attribute to *yes*:

$$i_1 \texttt{ / } 0 \xrightarrow{\{halting'=yes,\ldots\}} i_1 \texttt{ / } 0 \qquad \text{(MSOS-Div-By-Zero)}$$

$$\texttt{halt} \xrightarrow{\{halting'=yes,\ldots\}} \texttt{halt} \qquad \text{(MSOS-Halt)}$$

As desired, it is indeed the case now that a fact of the form $s \xrightarrow{\{halting'=yes,...\}} s'$ is derivable if and only if $s = s'$ and the next executable step in s is either a halt statement or a division-by-zero expression. The setting seems therefore perfect for adding a rule

$$\frac{s \xrightarrow{\{halting'=yes,...\}} s}{s \xrightarrow{\{halting'=\epsilon,...\}} \texttt{skip}}$$

and declare ourselves done, because now an abruptly terminated statement terminates just like any other statement, with a `skip` statement as result and with a label containing a non-halting status. Unfortunately, that does not work, because such a rule would interfere with other rules taking statement reductions as preconditions, for example with the first precondition of the (MSOS-Seq) rule, and thus hide the actual halting status of the precondition. The only apparent solution to properly capture the halting status is to define a top level construct like we did for the big-step and small-step semantics above, say "`pgm` *Stmt*", and then include the following two rules:

$$\frac{s \xrightarrow{\{halting'=\epsilon,...\}} s'}{\texttt{pgm}\ s \xrightarrow{\{halting'=\epsilon,...\}} \texttt{pgm}\ s'} \qquad \text{(MSOS-Pgm-Normal)}$$

$$\frac{s \xrightarrow{\{halting'=yes,...\}} s}{\texttt{pgm}\ s \xrightarrow{\{halting'=\epsilon,...\}} \texttt{pgm skip}} \qquad \text{(MSOS-Pgm-Halt)}$$

Note that, even though MSOS can be mechanically translated into SOS by associating to each MSOS attribute an SOS configuration component, the solution above to support abrupt termination modularly in MSOS is *not* modular when applied in SOS via the translation, because adding a new attribute in the label means adding a new configuration component, which already breaks the modularity of SOS. In other words, the MSOS trick above cannot be manually used in SOS to obtain a modular definition of abrupt termination in SOS.

Exercise 43. *Same as Exercise 41, but for MSOS instead of big-step.*

Reduction Semantics with Evaluation Contexts

Reduction semantics allows very elegant, natural and modular definitions of abrupt termination, without having to extent the syntax of the original language:

$$\langle c, \sigma, \omega \rangle[i \ / \ 0] \to \langle \texttt{skip}, \sigma, \omega \rangle \qquad \text{CxtRed-Div-By-Zero}$$

$$\langle c, \sigma, \omega \rangle[\texttt{halt}] \to \langle \texttt{skip}, \sigma, \omega \rangle \qquad \text{CxtRed-Halt}$$

Therefore, the particular evaluation context in which the abrupt termination is being generated, c, is simply discarded. This is not possible in any of the big-step, small-step or MSOS styles above, because in those style the evaluation context c is captured by the proof context, which, like in any logical system, cannot be simply discarded. The elegance of the two rules above suggest that having the possibility to explicitly match and change the evaluation context is a very powerful and convenient feature of a language semantic framework.

Exercise 44. *Same as Exercise 41, but for reduction semantics with evaluation contexts instead of big-step.*

The Chemical Abstract Machine

The CHAM semantics of abrupt termination is even more elegant and modular than that using context reduction above, because the other components of the configuration need not be mentioned:

$$\{\!|i \texttt{ / } 0 \triangleright c|\!\} \to \{\!|\texttt{skip}|\!\} \qquad \texttt{(CHAM-Div-By-Zero)}$$

$$\{\!|\texttt{halt} \triangleright c|\!\} \to \{\!|\texttt{skip}|\!\} \qquad \texttt{(CHAM-Div-By-Zero)}$$

Exercise 45. *Same as Exercise 41, but for the CHAM instead of big-step.*

3.7.4 Adding Dynamic Threads

Big-Step Operational Semantics

Small-Step Operational Semantics

Modular Structural Operational Semantics

Reduction Semantics with Evaluation Contexts

The Chemical Abstract Machine